

---

# Real-Time Design with Peer Tasks

---

Andre Goforth, Norman R. Howes,  
Jonathan D. Wood, and Michael J. Barnes

---

October 1995



National Aeronautics and  
Space Administration

---

# Real-Time Design with Peer Tasks

---

Andre Goforth, Ames Research Center, Moffett Field, California

Norman R. Howes and Jonathan D. Wood, Institute for Defense Analyses, Alexandria, Virginia

Michael J. Barnes, Recom Technologies, Moffett Field, California

October 1995



National Aeronautics and  
Space Administration

**Ames Research Center**

Moffett Field, California 94035-1000

# Real-Time Design with Peer Tasks

ANDRE GOFORTH, NORMAN R. HOWES,\* JONATHAN D. WOOD,\* AND MICHAEL J. BARNES†

*Ames Research Center*

## Summary

We introduce a real-time design methodology for large scale, distributed, parallel architecture, real-time systems (LDPARTS) that approaches system scheduling analysis in a way different from those methods that use a scalar metric of urgency such as found in rate (or deadline) monotonic theories. The latter assume the place for scheduling prioritization to be at the functional level of run-time processes. For example, in the Ada programming language this refers to task scheduling. In our method, the fundamental units of prioritization, which we call *work items*, are system level or domain specific objects with timing requirements (deadlines) associated with them in the requirements specification. For LDPARTS, a work item consists of a collection of tasks. No priorities are assigned to tasks or, equivalently, tasks have equal priorities. Such a collection of tasks is referred to as *peer tasks*. Current scheduling theories are applied with artifact deadlines introduced by the designer whereas our method schedules work items to meet specification deadlines (sometimes called end-to-end deadlines) required by the user.

The new method supports these scheduling properties. The scheduling of work items is based on domain specific importance rather than task level urgency and still meets as many work item deadlines as can be met by scheduling tasks with respect to urgency. Second, the minimum (closest) on-line deadline that can be guaranteed for a work item of highest importance, scheduled at run time, is approximately the inverse of the throughput, measured in work items per second. Third, throughput is not degraded during overload and instead of resorting to *task shedding* during overload, the designer can specify which work items to shed. We prove these properties in a mathematical model (ref. 1). The degree to which these hold for a specific system depends on how small the statistical variance of work item throughput is over the required system performance envelope. The method works best in a project with a “cut and try” iterative development approach, where measurement of work item throughput

may be made under as realistic system load conditions as possible.

## 1. Introduction

The real-time methodology we introduce addresses three issues in large scale, distributed, parallel architecture, real-time systems (LDPARTS). We base the feasibility of our approach on empirical data found in the application of our method to an existing case study. Though the case study is in the Ada83 programming language, which comes with its own run-time environment, we believe the method is readily applicable in any environment with inter-process communication (IPC) and network services that provide robust asynchronous concurrency (ref. 2). The three unique issues are how to:

- (1) do on-line scheduling with deadline guarantees;
- (2) minimize scheduling overhead; and
- (3) sustain performance in overload conditions.

The first issue has been discussed as the “disconnect” between real-time design theory and scheduling theory at a workshop on Large, Distributed, Parallel Architecture, Real-Time Systems (ref. 3). This workshop was held at Institute for Defense Analyses (IDA) and sponsored jointly by the NASA Ames Research Center and the Ballistic Missile Defense Office (BMDO).

Designers of complex, real-time systems must address robustness, efficiency, and availability as well as timing (ref. 4). In many cases, the resulting system decomposition is not readily, if at all, analyzable by current scheduling theory techniques, hence the disconnect. While workshop participants agreed that real-time systems need to have correct timing (i.e., meet required deadlines), they disagreed about systems being designed to accommodate a specific scheduling approach.

All participants agreed that schedulability analysis needed to be part of the design process. On one hand, the scheduling theorists thought that schedulability analysis should drive the design process. Their argument was that, given the ability of a real-time system to meet its timing requirements is so important, if schedulability analysis (often referred to as feasibility analysis) is not begun in

---

\*Institute for Defense Analyses, Alexandria, Virginia.

†Recom Technologies, Moffett Field, California.

the earliest phases of design, the resultant system may not be capable of meeting its timing requirements. On the other hand, several designers were not convinced that, even if the system's tasks satisfied the constraints of a given scheduling theory, this actually ensured that the final system would behave as predicted. One reason given is that current scheduling theories do not adequately make allowances for all significant factors related to "real world" behavior found in LDPARTS.

In regard to the second issue, some participants noted that current scheduling theories are applied to artifact deadlines, introduced by system designers, and not to specification deadlines, required by the users, thus introducing, arguably, significant overhead, just to be able to use these theories. Rate monotonic theory (ref. 5) is one example that was cited, especially in attempts to use it in LDPARTS.

The third issue, which was not as universally important to workshop participants as the first two, was very important in the kind of systems ARPA (Advanced Research Projects Agency), our sponsor, had us investigate. In section 3, we discuss a hypothetical air defense system where, because of the hostile environment, so-called nominal operating conditions must include sustained operation in overload conditions. Put another way, such systems must be designed to operate in overload conditions as if these were part of nominal operating requirements.

Our design approach consists of the application of these fundamental rules:

*Rule 1. Schedule work items by importance to meet their specification deadlines. Use peer task sets to accomplish these work items.*

*Rule 2. Model each real world process with a single task (independent thread of control).*

*Rule 3. Reduce the mean service time of cyclic functions.*

The underlying concepts and rationale for these rules and how to apply them are discussed in section 4.

Our approach addresses the three issues by introducing the paradigm of *work item* and its *importance* as a metric for scheduling prioritization. In contrast, some real-time scheduling approaches are based entirely on *urgency*, such as Earliest Deadline First (EDF) where the next task to be executed is the one with the nearest deadline. Any task significance is identical to the task's deadline, i.e., urgency. For real-time problems where all or most of the information necessary to schedule the tasks is known in advance, scheduling based on urgency can be appropriate. The reason is because the designer of the system may be

able to size the system appropriately based on a priori knowledge so as to rule out the possibility of ever having the system in a state of "overload," i.e., a state in which there are more requests for real-time service during some (possibly temporary) period than the system can possibly respond to. Such is often the case for closed loop control systems and low level sample data systems. The concept of *importance* is elaborated on in section 2.

Another element of our approach is its focus on work item throughput. A common design characteristic of real-time systems is throughput. The required throughput or rate, items processed per second, depends upon the needs of the application. For many applications real-time behavior means "fast enough." For others, there is an additional requirement of a sufficiently small variance in the throughput. For example, consider the case of a system, X, that can accomplish tasks at some fixed rate, say 30 milliseconds, with a very small standard deviation, and another system, Y, that can accomplish the same tasks at some fixed rate, say 25 milliseconds, but with a large standard deviation. Now Y is clearly "faster" than X, but if we find the standard deviation of Y to be so large that it would occasionally cause Y to perform the task in, say 75 milliseconds, whereas the standard deviation of X is so small that it could never (in the lifetime of the system) execute the task in over 31 milliseconds, then we may claim that X exhibits "real-time behavior" with guaranteed deadlines, while Y does not.

In our experience with the method which is discussed in section 5, we have found work item performance to be sufficiently stable so as to allow us to draw conclusions about the system's ability to meet hard deadlines. Furthermore, work item throughput was found, for all practical purposes, to be constant even in overload conditions. The scheduling properties supported by our methodology are as follows:

(1) Work item processing time (after preemption of any lower importance work item) is essentially constant, and the inverse of the throughput, measured in work items per second, can be considered the minimum on-line guarantee that can be met if no work item of higher or equal importance awaits processing.

(2) Work items based on importance may be scheduled without incurring degraded throughput caused by on-line task scheduling.

(3) The maximum number of the most important work item deadlines may be met during overload and instead of resorting to task shedding during overload, the designer can specify which work items to shed.

We provide proofs of these in section 6. We use the mathematical model and results of Detouzos and Mok

(ref. 1), which we will refer to as the D-M model. It is common knowledge that when scheduling theories are applied to real world problems, the results are not what the theories predict. Why is this, in view of the fact that these scheduling theories are “proved mathematically”? It is because mathematical models, including the D-M model, make a number of assumptions about the real-time environment so as to render them not sufficiently accurate in predicting how a real-time system will behave. Some such assumptions are (1) there is no scheduling overhead, (2) there is no preemption overhead, (3) the computation time of each task is constant, (4) the tasks are relatively independent, at least from a scheduling point of view, and (5) time is quantized rather than continuous. This does not mean these models are of no use; rather, their predictive power is limited to certain environments. Only careful, comprehensive testing can assure that real-time systems meet their deadlines. The “guarantees” of real-time scheduling theories are helpful insofar as careful, comprehensive testing can verify how well the system behaves in relation to the theory as well as, and more importantly, in relation to the requirements. In section 5, we discuss our experimental findings of real-time behavior that matches what the theory predicts, properties 1 through 3 above, and behavior that deviates from the theory.

We extend our appreciation to Mark Boyd and David Galant of Ames Research Center for their diligent efforts in reviewing this paper and making significant recommendations.

## 2. Significance and Urgency

For our purposes, the concept of *significance* has to do with the impact of what will happen if a work item misses its deadline. The concept of *urgency* has to do with how close the deadline for the work item is to the current time (time now). It is possible to have work items that are highly significant but not urgent or that are very urgent but not significant. In general, all work items have some measure of significance and urgency, and there may be work items that are simultaneously very significant and very urgent. Our concept of importance is the primitive idea of doing the right thing at each instant, in the best interest of providing the desired solution to the problem. The domain expert or “problem owner” is in the best position to identify importance. At the workshop several real-time specialists stressed the need to have real-time systems that do not give priority to executing significant tasks that are not urgent in favor of urgent tasks that may not be significant, thereby causing a deadline for a (possibly insignificant) task to be missed unnecessarily. The approach we advocate does not violate this need.

This follows from our concept of importance as a combination of significance and urgency. Formally, we write importance as a function of significance and urgency, i.e.,  $I = F(s, u)$  where  $F$  is some function of the significance  $s$  and the urgency  $u$ . Furthermore, importance is a dynamic concept in that its value may be a function of time, where both significance and urgency are functions of time, say  $s(t)$  and  $u(t)$ . Clearly, the urgency  $u(t)$  is continuously changing with time, while the significance  $s(t)$  may be constant or changing with time. We may write  $I$  as  $I(t) = F[s(t), u(t)]$ .

For our purposes, we assume that  $F$  is definable so that urgent work items will not miss their deadlines unnecessarily, i.e., be missed while a work item of higher importance is executed whose urgency is low enough (deadline far enough away) that the urgent work item could have been executed and still have had enough time to meet the higher importance work item’s deadline. We make no attempt to define a particular importance function  $F$  for each work item because such functions are domain specific. We assume that these functions exist and can be evaluated at any time  $t$ . Furthermore, we will not deal with the issue of how long it takes to compute the importance functions on line. One can construct pathological problems in which the time to compute the importance functions is arbitrarily large. However, for many important problems the time to evaluate the importance functions is bounded and small. Clearly, an extremely complex importance function may not be suitable for a given real-time problem. How to design importance functions is not within the scope of this paper. What we will discuss in the next section is how to use them under the assumption that appropriate importance functions can be evaluated at any time  $t$ . As an example, note that “aging algorithms” can be incorporated as  $u(t)$  in an importance function  $F$  so that the importance of a task becomes greater as time passes.

## 3. LDPARTS Example

A real-time problem (ref. 6) that illustrates the class of problems our method addresses is as follows: Imagine an air defense system that must be capable of simultaneously handling up to 1,000 “radar tracks” of possible “threats (targets),” up to 100 known threats (tracks that have previously been identified as targets), and up to 10 engagements where an engagement means that a defensive weapon system is employed against a known target. Tracks may include many things other than threats, such as civilian aircraft in the area, friendly aircraft, decoys, and birds, as well as real threats.

The numbers 1,000, 100, and 10 are used only for illustration. It is not meant to imply that a real air defense system would have this requirement. The differences in orders of magnitude from 10 to 100 to 1,000 are one example of a physical characteristic typical of an LDPART system. For instance, the air defense system may only have 10 launchers or 10 anti-aircraft weapons with which to engage targets. The amount of processing may increase as targets become known and approach the defended area more closely, thereby limiting the number of actual targets that the system can handle to only 100.

Suppose that the radar subsystem, because of its physical characteristics, gets information on tracks once a second but that tracks can appear aperiodically on the radar screen and also vanish aperiodically from the radar screen. Further, suppose there is a requirement that once a track “appears” on the radar screen, it must be determined within 2 seconds whether the track is a target. The test or tests that are performed to determine this may not be conclusive, in which case the track may appear again to the radar within another second and need to be reprocessed to determine whether the track is a target. This goes on until the track either disappears from the screen or possibly gets so close that it is automatically classified as a target.

Furthermore, suppose that once a track is classified as a target, it then has to be “monitored” within 1 second each time the radar “illuminates” the target. All targets get illuminated approximately once a second, but since the targets are moving with respect to the radar, the time between illumination need not be periodic for an individual target. The monitoring of the track determines the relative lethality of the target and therefore influences the significance of the target. Finally, suppose that once a target is identified it must be engaged within 2 seconds provided there is a weapon system available to engage it, with priority being given to engaging the most important threats first. Suppose further that for the targets that are currently engaged, guidance update information has to be relayed to the defensive weapon (say an air defense missile) every 100 milliseconds while it is homing in on the target.

For each requirement corresponding to the various deadlines of 2 seconds, 1 second, and 100 milliseconds, there is a significance associated with the requirement. If the calculation associated with one of the 1,000 tracks misses its deadline, the track may temporarily be “lost.” This has to do with the fact that tracks frequently “cross” on the display corresponding to points in time where the radar cannot distinguish between two tracks and therefore may not know, for instance, what direction the track is

moving in without doing a “correlation” calculation. If a track is “lost” it may quickly reappear and no overall harm may be done as the calculations can begin again shortly thereafter. Though there is some risk associated with this momentary loss because precious time is used in updating track information, we may assume that the significance of updating an unidentified track, in these circumstances, is small.

However, if a track that has been already classified as a target is temporarily lost, the risk is greater. Furthermore, not all threats have the same lethality. One incoming target may have the capability to wipe out the whole air defense system while another may only have the capability to degrade its performance. The air defense system is also responsible for protecting other assets in the area as well as itself. Some identified targets may not be approaching the air defense system but moving toward one of these assets. The value of the asset then affects the significance of the target. Finally, if a target is engaged and the guidance update deadline is missed, the defensive weapon may be temporarily or permanently lost, which may be the highest risk of all.

To complicate the situation, suppose that it is possible for the enemy to field thousands of decoys, or the system may have to be used in the “heat of battle” where the friendly aircraft in the vicinity together with civilian traffic and enemy threats temporarily exceed the capacity of the system in one way or another, i.e., more threats than 100 or more engageable targets than 10. Air defense systems are often deployed in multiple locations in an area. They are designed to share the overall battle load, but due to chance, one air defense unit may encounter much more of the load than others. It is therefore impossible to design such a system so that it will never encounter overload.

It may be becoming clear to the reader that if the scheduling of tasks is based only on urgency, we may maximize the number of deadlines met while unnecessarily missing the really important work item deadlines. There are two things left to discuss about this example. First we want to discuss why such systems tend to start missing critical deadlines prematurely, and second we want to discuss the behavior of current scheduling algorithms during overload. Both of these discussions provide motivation for the method we will introduce.

First, real weapon systems of the type suggested by our example are often geographically distributed and so complex that multiple designers are required to design various parts of the system. There are literally hundreds of tasks (independent threads of control) and no one person understands all the ramifications of task-to-task

interaction. Because the current mathematical models used in real-time analysis do not sufficiently account for all the scheduling factors in the real world problem space, these systems often begin missing deadlines at loads far lower than the theoretical overload level for which they were designed.

Why do such systems tend to start missing critical deadlines prematurely? In our example, note that although the physical constraints of the problem (namely the behavior of the radar) tend to give the problem a periodic flavor, the problem is fundamentally aperiodic because radar tracks “appear” at random times. After first appearance, if the track is not lost, the radar updates tend to be fairly periodic if the target is moving slowly or if it is approaching. But not all targets satisfy these two requirements. Also, threats are identified aperiodically. Past attempts, rooted in using periodicity as a system organizational principle, try to solve problems like this by assigning a single task the job of doing the computation for each of these mostly periodic functions, e.g., having a single task do the computation for updating all the tracks. A rationale, for example, is that all the radar updates for all the tracks will be obtained within approximately 1 second, leaving one more second in which to do the computations, without missing the deadline for any of the tracks. At the end of 1 second, another set of updates will become available for the next execution of the task.

The difficulty with this design approach is that the individual tracks are the items that have significance. If a single task is doing the computation for all the tracks, it is mixing the most significant work with the least significant work without discriminating one from the other. An alternative and preferable approach is to categorize tracks by their significance as they are identified and give priority to the processing of more significant tracks as the load increases. Friendly tracks have to be maintained as well as enemy tracks during battle, but under all load conditions—nominal, significant, or overload—there needs to be a way of emphasizing the threats and disposing of them. On the other hand, we do not have to go to the other extreme of assigning one task per track. An effective design allows for assignment of end-to-end resources that process according to significance or importance as identified in end user requirements. The emphasis on decomposing the system workload into periodic components and scheduling them based on urgency or frequency tends to obscure this importance and, particularly, how the system will behave when

processor utilization is high, i.e., in periods of heavy load or overload. As Briand (ref. 7) states in his paper:

*It is not because a task's execution time is the shortest or that its deadline is the closest that is more important [but] that this task preferably achieves its deadlines. We are driven back to the initial problem of the task priority allocation algorithms: the only way to formulate a non-specific algorithm is to ignore the task's semantics, to consider that all of them are created equal, though they are not. Such a priority allocation algorithm is valid only if from an applicative point of view, no task is more important than another (no program feature is in jeopardy when the concerned task can't execute within its normal execution window). In the real computing world, there is often a known risk of processing resource shortage during heavy load situations. Specifically in those situations that are irrelevant with the common scheduling model paradigms, one should carefully balance the applicative consequences of missing a deadline before mechanically allocating priorities.*

For our example problem, dealing with a threat (target) as a whole is important so as to be able to meet the overall deadline of engaging and destroying the threat before it damages us. Meeting all the component deadlines of all the tasks that cooperate together to do the computations necessary to effect this end objective must be considered secondary. More important is identification of a new track as a target and to engage it within 4 seconds if at all possible, for example. This is the essential deadline and the only one that really matters in the final analysis. There may be scores of tasks that meet 10 millisecond artifact deadlines to accomplish this feat; it really does not matter. What matters is meeting the 4 second specification deadline specified in the requirements.

The second point is that if real-time air defense systems or battle management systems rely on urgency-based scheduling techniques alone they may not behave as planned under heavy load. Clark (ref. 8), in a paper done for Rome Air Development Center, demonstrates how several widely used scheduling policies lose their ability to meet deadlines as they approach and exceed 100 percent load. This results in the well known throughput decay experienced in many real-time systems as load increases. Figure 1 depicts his results for sets of tasks that do not share resources. The results get worse as the tasks become less independent (i.e., cooperate more).

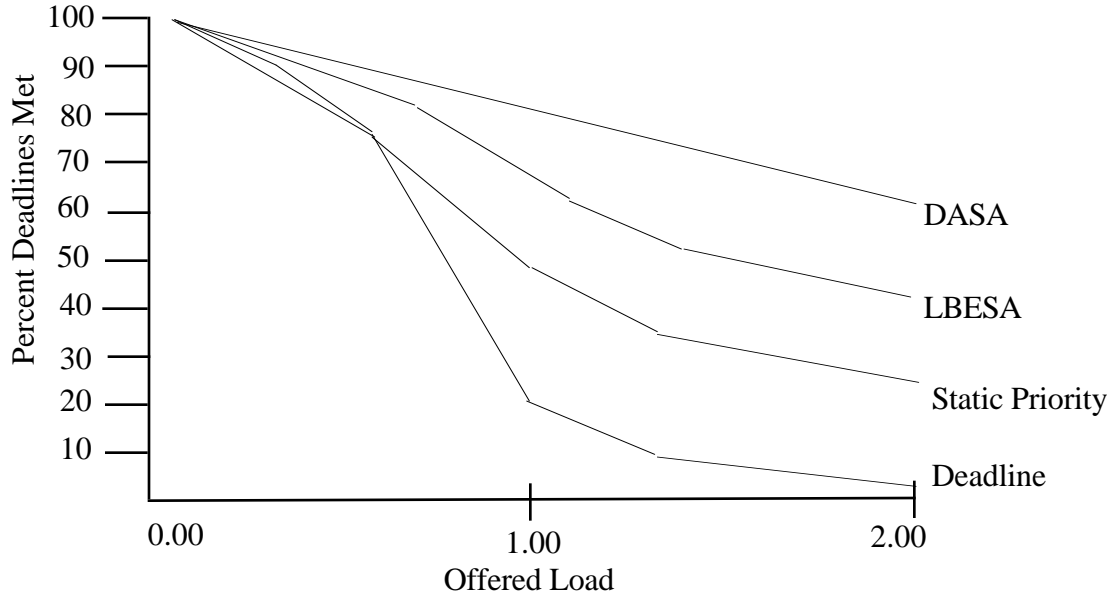


Figure 1. Percent deadlines met.

The four curves in figure 1 represent the Dependent Activity Scheduling Algorithm (DASA) introduced by Clark in the paper, Locke's Best Effort Scheduling Algorithm (LBESA), a static priority scheduling algorithm, and the Earliest Deadline First algorithm which he labels as "Deadline" rather than EDF. Notice that EDF does the worst under load and yet it is the algorithm that has been proven to be "optimal" in the D-M model (for the single processor case) in the sense that if any task set can be scheduled by any other algorithm to meet some set of deadlines, then it can be scheduled using EDF to meet those deadlines. This serves to illustrate the difference between the mathematical models in which scheduling theorists work and the real world. Briand (ref. 7) also presents some similar graphical information showing how urgency based scheduling can lead to premature missing of deadlines. Even though in overload conditions Clark's DASA algorithm meets more deadlines than the other algorithms, these may not be the most important deadlines. Clark also shows that his algorithm "accumulates more value" in the time value sense than the other algorithms thereby indicating that it is better at distinguishing significance while still meeting the most deadlines. This makes Clark's algorithm an attractive contender for LDPARTS. But Clark's algorithm does not ensure that the most important work will get done at any instance in time. Our approach addresses this concern and addresses the concern of minimizing throughput degradation during overload.

#### 4. The Peer Tasking Design Method

The peer tasking design method derives its name from the fact that the tasks in the system are not prioritized. If tasks have no priorities, how can important functions get done ahead of functions of lesser importance? The key idea is that the work is scheduled at a higher level of granularity than individual tasks.

The concept of a *task* is used in different ways in the real-time literature. Often it means some schedulable unit of work. At other times, it means an individual thread of control wherein multiple tasks cooperate. We will need both of these distinct concepts and will try to make clear in which sense we are using the term task. Most real world, real-time system designs decompose real-time requirements from the system specification into individual processes called "tasks" and assign "deadlines" to these tasks, while the real-time specification provides deadlines with respect to the requirements rather than with respect to these individual tasks. We refer to these individual task deadlines as *artifact deadlines* because they are artifacts of how the designer decomposed the system into tasks rather than any inherent timing requirement in the specification.

To be precise, we will refer to the deadlines called out in the requirements specification as *specification deadlines* to distinguish them from artifact deadlines. From many discussions with real-time practitioners at conferences

and workshops and from observing how certain government systems were designed, we believe that common practice takes a requirement from the specification that has a deadline associated with it (i.e., a timing requirement) and refer to that deadline as the “time budget.” That requirement is then decomposed into individual “tasks” that may either be smaller components of work that can be parceled out to programmers, or concurrent threads of control in a concurrent design, or some mixture of both. A sequential order in which these tasks have to execute or a directed graph structure that determines an ordering in which they can execute may or may not exist. The time budget is then broken down into individual time budgets for the individual tasks that the designer believes will ensure that the overall requirement will be completed within the specified deadline, if each of the individual tasks meets its time budget.

We refer to these work requirements in the specification that have associated deadlines as work items. These are the units of work that we will schedule (by the application program—not the operating system or the language run-time system) rather than the individual tasks that accomplish the work items. Work specified in the requirements specification with no associated timing requirements will be executed in the background as time permits. These lower level (finer granularity) units of work that we are calling tasks have no intrinsic priority, but while they are executing, they can be thought of as inheriting the priority of the work item. It is not necessary, however, to think of them as having any priority whatsoever, because it adds nothing to the design. Thus, a work item is a collection of tasks, which may or may not have a linear or partial ordering, that performs the work item’s execution.

Work items may be independent or they may cooperate to accomplish a larger objective. We will not require that they be independent. Instead, we require that a higher importance work item preempt a lower importance work item. We also assume that specification deadlines are associated with work items and, if work items cooperate to perform larger objectives, then there are no deadlines associated with these larger objectives. In our air defense example, the work items may be the engagements of individual threats and these work items cooperate in the overall objective of air defense, but there is no deadline associated with air defense. This brings us to our first design rule:

*Rule 1. Schedule work items by importance to meet their specification deadlines. Use peer task sets to accomplish these work items.*

This rule implies that no work item will miss its deadline unnecessarily. A lower importance work item  $w_1$  misses its deadline, only because some work item  $w_2$  of equal or

higher importance was executed instead. Since  $w_1$  could not also be scheduled to meet its deadline at the time  $w_1$  needed to begin execution to meet its deadline, the system was in an overload state. Consequently, it was necessary to shed  $w_1$  in favor of  $w_2$ . In an overload state the system must shed some of the load. In our approach, load shedding is based on importance. Of course the shedded load may be temporarily buffered in hope that the overload condition is temporary, and where there is still importance associated with doing certain work items late. With current real-time scheduling algorithms, work shedding is done at the task level and is based on some other criteria, e.g., urgency or frequency.

We have discussed the decomposition of the system work load into work items for scheduling purposes, but we have not discussed decomposing the system into individual tasks as part of the design process. Based on the discussion so far, one might think that the way to proceed would be to decompose work items into tasks. There is no requirement that each work item have a separate task set.

We propose that the decomposition of the system into tasks is done according to what we call *process modeling*, a design technique referred to as *physical concurrency* in reference 9. In this context, process means a real world process in the problem domain as opposed to a process in the sense of an executable unit of code. A real world process is a set of coordinated activities that accomplishes some larger function. (In the language of object oriented design a process is a group of cooperating objects.) A word of caution: work items are not processes. Work items are a decomposition of the system workload. What we are describing here is a decomposition of the system itself. In the air defense example, managing the input stream of radar tracks—which includes interrupt handling, segmenting the incoming stream into individual tracks, and buffering of track updates—is a real world process. Correlating track updates to existing tracks is another. Processing an unclassified track to determine if it is a target is yet another, as is processing an identified target to determine its lethality (significance). By process modeling we mean modeling each of these real world processes in the designed system with a single task (independent thread of control). This brings us to our second rule:

*Rule 2. Model each real world process with a single task (independent thread of control).*

The central idea in Rule 2 is that process modeling should be the only use for concurrency (application level processes or tasks). The specification of semaphores, buffering messages, and other “low-level” concurrency constructs should not part of the application or global level design.

The partitioning of the system into tasks using Rule 2 produces a design in which all tasks cooperate to produce one or all of the work items. A consequence is that no task can be shed during an overload condition because no work items could then be accomplished. This characteristic will be evident in the design used in our case study.

Nothing in the rule requires one task to have priority over another. Without priorities, task scheduling defaults to merely a first come first serve dispatcher. Such is the case with Ada83 tasks (and, in our case, Ada running on top of Unix operating systems). Furthermore, nothing in the rule requires synchronous process or task interaction, which, again, is the case with Ada 83 tasks.

Finally, we have the issue of the design optimization. The rules we have advanced thus far may be thought of as architectural structuring principles with first and second order effects on performance. Once the architecture is defined, optimization may provide a higher order effect. Our approach to optimization tunes at the intertask and task interface level.

The real world processes in the problem are being modeled by individual tasks (independent threads of control) in our approach. If we think of these tasks as an interrelated collection of queues we see that the only way to increase throughput (performance) is to reduce the mean service times of these queues. For a given queue, two things contribute to its mean service time. The first is the time to service the queue, and the second is the time to get a request from an “upstream” queue to this queue. If the implementation language is Ada83, the choices of how the tasks interface (e.g., conditional entry calls as opposed to a bare rendezvous) influence the latter delay so it is often helpful to experiment with various task-to-task interface methods. Continuing this interrelated collection of queues analogy, we see that within the individual tasks there are functions performed repeatedly which require significant computation time. We will refer to these as *cyclic functions*. If we think of these cyclic functions as queues, we have collections of queues within queues. To increase throughput, then, requires reducing the mean service times of these cyclic functions, although a given time decrease may not lead to a decrease at the work item level.

To systematically reduce the mean service time of cyclic functions, one must first learn what these service times are. This presupposes that we can measure these service times which, in turn, implies that there is something to measure. Consequently, we advocate the following development approach.

Design the system using Rules 1 and 2. Then implement the real-time components of the system, stubbing off non-

real-time components. Then measure the throughput of work items in the various scenarios of interest. This gives us the deadlines that can be “guaranteed on-line” by the system along with the standard deviations associated with these guarantees. If any of these throughputs, deadlines, or standard deviations fall short of our expectations, we make further measurements, but only in the areas that might affect these throughputs or deadlines.

The next round of testing measures the mean service times of the cyclic functions. To perform these measurements, an event recording package or a monitoring tool is helpful. Knowing the mean service times of the cyclic functions provides insight as to where the bottlenecks reside. This process is continued until it can be determined that the system can meet the required specification level deadlines, or an iteration of Rule 1 or 2 or both are needed. We summarize the process in the following rule that we call the *tuning rule*.

*Rule 3. Reduce the mean service time of cyclic functions.*

Designing in accordance with these three rules we call the *peer task (process) design method*.

## 5. Case Study

The Remote Temperature Sensor (RTS) system is an example of a real-time system dedicated to the monitoring of a physical system (in this case, 16 furnaces). The RTS example originally appeared in the book titled *Real-Time Languages: Design and Development* by Young (ref. 10). Nielsen and Shumate present a version programmed in the Ada language as Case Study No. 1 (ref. 11). Sanden (ref. 12) and Howes (ref. 13) provide additional studies that include critiques of Nielsen and Shumate’s design. In figure 2, the Howes design is presented via Buhr diagrams (ref. 14) and served as a basis for the case study discussed below.

We provide a short summary of the Buhr diagram notation used here and refer the reader to the above reference for an in-depth explanation. The parallelograms denote Ada tasks which are concurrent objects or processes. Interconnecting lines denote communication, which may be thought of as asynchronously occurring, concurrent procedure calls. In these calls, parameters may be passed in either direction; the direction is usually denoted with an arrow, and the passed parameter values, if any, are denoted next to the line. Ada provides alternatives for making calls that are conditional, i.e., that depend on the readiness of the called or the length of time one is willing to wait for a call to be accepted. These are denoted by arcs that approach a parallelogram (task) and

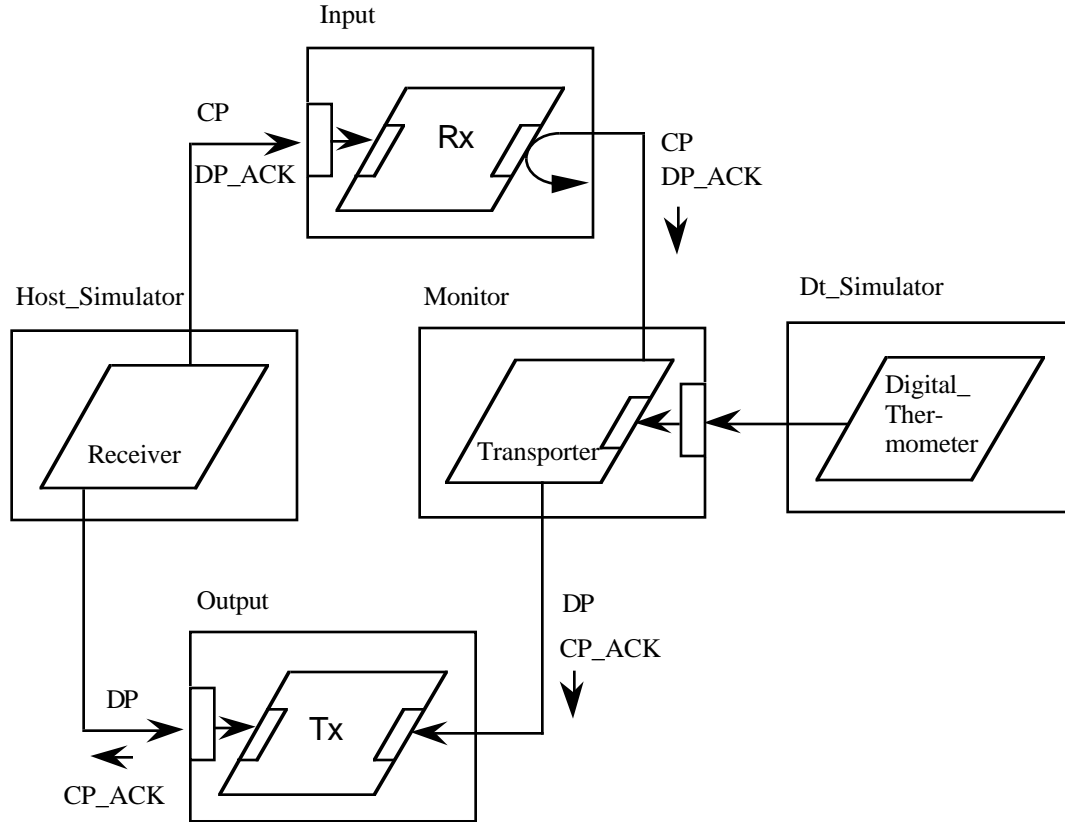


Figure 2. Howes design.

curve back toward their origination. Some of the diagrams show parallelograms contained within larger boxes and show arrows touching very small boxes tangent to the sides of the larger boxes. These denotations deal with Ada specific details where the designer has chosen to guard the accessibility of objects according to acceptable software engineering practice.

Briefly, what the RTS system does is receive messages from a host system in the form of control packets (CPs) and send messages back to the host in the form of data packets (DPs). Each DP contains a furnace number and its temperature. These temperature readings are obtained by sending a furnace number to the digital thermometer, which returns that furnace's temperature. The CPs from the host are used to change the rate at which the furnaces are read. They contain a furnace number and its sensing rate. In our design, we allowed for the requirement that different furnaces may be sampled at different and unrelated rates and that these are subject to change at any time with the issue of a new CP from the host.

This is an example of a system with dynamically changing sensing rates that makes designing to accommodate a scheduling policy based on static, or at most finitely many

different, periodicity sets (modes) unwieldy. Note also that the processing of periodic and aperiodic messages is interwoven which further compounds the application of such a scheduling policy. The arrival rate of *control packets*, *control packet acknowledgments*, and *data packet acknowledgments* is random, whereas the arrival of *data packets*, which are generated at the current sensing rate for a given furnace, is periodic.

Our design of the RTS system divided the work load of the system into two work items, namely the processing of a CP (CP work item) and the processing of a DP (DP work item). CP processing involves (1) the receiving of a CP message from the host system, (2) the updating of the furnace reading database, (3) the packaging of a CP Acknowledgment message, and (4) the transmission of this message to the host. DP work item processing involves (1) monitoring the furnace reading database and the clock to determine if it is time to read another furnace, (2) sending a furnace number to the digital thermometer, (3) receiving a temperature reading from the digital thermometer, (4) packaging the temperature reading into a DP, (5) sending the DP to the host, (6) receiving the DP by the host, and (7) receiving the DP Acknowledgment from the host system.

We now discuss measurements made to verify how well the case study's real-time behavior matches the three scheduling properties mentioned earlier. These measurements are by no means exhaustive, but do illustrate the feasibility of our approach.

Since the processing of DPs (in the absence of CPs) is essentially periodic, we first measured the minimum on-line deadlines that could be met for a processing load consisting solely of DP work items. We also measured the timing stability; i.e., the standard deviation of work item throughput obtainable with our design.

We determined that the mean time for processing a DP work item on a SUN 3/60 was 7.348 milliseconds and the standard deviation was 37.4 microseconds. In other words, there was an extremely small variance around the mean and an on-line guarantee of, say, 8 milliseconds could be made that would be practical for a reasonable design. If a more conservative guarantee was desired, then statistically an on-line guarantee of 9 milliseconds would not be violated within the lifetime of the system. The mean and standard deviation of CP work item requests for service were found to be 9.915 milliseconds and 129.6 microseconds. This timing stability (at the work item level) is a step toward verification of scheduling property 1.

Next, we measured the mean and standard deviation of DP work item processing during heavy overload. It was found that the behavior was exactly the same and likewise for CP work items. In our design, during overload, excess requests for service are shed and they are not buffered for later processing. During overload and non-overload, the DP work item requests for service are honored, by application program design, in a way that is fair with respect to all the furnaces, with the oldest request for service for a given furnace being shed first. We note that such timing stability was achieved with a general purpose operating system, Unix, and that other applications may achieve similar real-time behavior without resorting to the use of a special purpose real-time one. This result is a step toward verification of scheduling property 3.

We now report on measurements to verify whether work item performance incurs degradation in throughput caused by on-line task scheduling, which is scheduling property 2. In regard to this property, we found preemption of tasks (processes) to have a significant impact on the real-time behavior. Before we discuss our findings we discuss the issues of preemption in the D-M model and in implementation.

The D-M model assumes that preemption time is zero, so it has no predictive power when it comes to questions regarding preemption times in real-world systems.

Several real-time practitioners with whom we discussed preemption overhead were aware that, in their systems, preemption overhead is significant. Others seemed to ignore it based on the assumption that context switching times are "small" (usually in the 150 to 200 microsecond range) and can be included in the task's computation time for purposes of analysis. Clearly no real world preemption scheme can work perfectly, like the zero preemption assumption of the D-M model. In the real world we usually, if not always, have preemption points. Checks to determine if preemption is needed usually occur at certain points in the application program or, in the case of interrupt driven preemption, at regular intervals determined by signals from an interval timer. In the limit, it is conceivable that a system could be designed that would check to see if preemption was needed after each instruction was executed, but clearly this is an impractical extreme. In both cases, no preemption is possible between preemption points. It is also conceivable that a totally aperiodic system would only check to see if preemption was necessary when an external interrupt was received, but even in this case, preemption cannot occur when the system is in certain states, for instance, while a running program is in a critical section.

Consequently, receipt of a higher priority request for service while a lower priority item is being processed does not mean the higher priority request is serviced immediately. Some time will elapse before the higher priority request is honored. Conceptually, this time interval can be made arbitrarily small, but in the real world, doing so is often impractical (prohibitively expensive).

When scheduling work items instead of individual tasks, the question about preemption points takes new meaning. With fewer units of work (i.e., fewer work items than tasks) we should have lower scheduling overhead and lower preemption overhead. In many cases, it may be entirely adequate to preempt only at task execution boundaries since an individual task execution may only account for a small fraction of the time it takes to process a work item. The advantage of preempting only at task boundaries is that it can be accomplished at the application program level without preempting a running task. Thus the need for a preemptable kernel is eliminated.

To study the impact of preemption, we had to have a point of comparison, namely the case of no preemption. From previous measurements, we knew how long it took to process a work item of each type in the absence of work items of the other type, i.e., the default case of no preemption. Now, we studied the case of no preemption, where both types of work items were assigned the same

importance, and requests for service for both of these are made simultaneously and over a range of loads.

The application program's control logic (monitor package in fig. 2) ensured that a request for a DP work item is never made until the previous DP work item request is processed and likewise for CP work item requests. Hence only CP work item requests can be received during DP work item processing and vice versa. In an overload condition where there is an adequate number of requests of both types, this leads to an interleaving of CP and DP work item requests. Consequently, from what we already observed, we would expect that we could always guarantee that a CP work item could be processed within 17.263 milliseconds (the sum of the time to process first a DP work item and then a CP work item if both CP and DP work items have the same importance). We found that this is not the case. We found that there is something we call *contention overhead*. When requests for both CP work items and DP work items are constant and neither has priority over the other, there can be up to a 14.45 percent throughput degradation on a SUN 3/60. Consequently, the on-line guarantee has to be increased by 14.45 percent to at least 19.75 milliseconds, yet the timing stability of the system has not changed. We can still offer this guarantee with a similarly small standard deviation; rather, the guarantee is now not what the theory predicts, i.e., scheduling property 1. While this deviation may be an artifact of our design, it is possible that other phenomena are present in the Ada run-time system—phenomena, which manifest themselves in the real world, that are not taken into consideration in the D-M model.

Table 1 shows the results of an experiment we conducted that exhibits these unpredicted phenomena. In the experiment, we set the furnace reading frequencies so high that there would always be another DP work item to service before the previous DP work item was completed. We varied the number of CP work items from 1,001 to 12,001 in increments of 1,000 CP work item requests. These requests were equally spaced during the 2 minute (120,000 millisecond) run time for each individual experiment, rather than aperiodically. The meanings of the columns in table 1 are as follows. Column 1 is the number of CP work item requests during each 2 minute run. Column 2 is the actual number of CP work items processed. Column 3 is mean time to service a CP work item in the absence of DP work item processing. Column 4 is the actual number of DP work items processed (in between CP work item processing). Column 5 is the mean time to service a DP work item in the absence of CP work item processing. Column 6 is the total time that *should* have elapsed given the number of actual CP and DP work items that were processed and their respective mean service times.

Notice that, in the case of 1,001 CP work item requests, this total time figure is 126,207 milliseconds when in fact the run time was only 120,000 milliseconds (2 minutes). According to our theory, this should not be possible. Thus, when relatively small numbers of CP work item requests are intermixed with DP work item requests, the system *appears* to speed up. However, this phenomenon is not as mysterious as it first appears, but has to do with the default task scheduling we accepted from the Ada run-time system we used. When measuring work item performance on a single processor machine, it

Table 1. Result of equal priority work items with different deadlines

CP requests	Actual CPs	CP mean	DPs	DP mean	Total time	Percent	Contention overhead
1,001	1,001	9.915	15,825	7.348	126,207	105.00	-5.00
2,001	2,001	9.915	13,617	7.348	119,897	99.99	0.01
3,001	3,001	9.915	11,120	7.348	111,465	92.88	7.22
4,001	4,001	9.915	9,357	7.348	108,464	90.35	9.65
5,001	5,001	9.915	7,617	7.348	105,554	87.96	12.04
6,001	5,941	9.915	5,942	7.348	102,566	85.46	14.53
7,001	5,940	9.915	5,943	7.348	102,564	85.47	14.53
8,001	5,940	9.915	5,943	7.348	102,564	85.47	14.53
9,001	5,941	9.915	5,942	7.348	102,566	85.46	14.53
10,001	5,940	9.915	5,943	7.348	102,564	85.47	14.53
11,001	5,939	9.915	5,942	7.348	102,546	85.45	14.53
12,001	5,938	9.915	5,941	7.348	102,528	85.44	14.53

is necessary to run both the external event simulator and the RTS system on the same processor. CP work item requests for service are initiated by the host processor that is part of the external event simulator whereas DP work item requests are initiated in the RTS system itself. At the completion of a DP work item the default task scheduler in the Ada run-time system often schedules the tasks from the external event simulator next. If there is a CP work item request waiting, it can be initiated more rapidly than waiting until the transporter task runs again to start another DP work item. Consequently, we end up achieving even better throughput than we might expect because the CP and DP work item processing times included this additional task scheduling overhead in their original measurements. We discovered these findings by analyzing detailed event traces of similar runs.

As the number of CP work item requests increases, this apparent speedup disappears and gradually turns into a slowdown that culminates at a 14.53 percent decline in throughput over what the model predicts. When 6001 CP work item requests are made during the 2 minute run, the RTS system can no longer get them all done because CP work items and DP work items both have the same priority, and the system can only process about 5,940 of each during a 2 minute run. Therefore, with yet higher CP work item requests for service during a 2 minute run, the results do not change as table 1 shows. Because the experimental data produce a covariance of approximately zero, the CP and DP work item processing is essentially independent. At the present time we do not understand what causes this slowdown because the cause of the phenomenon appears to be below the threshold of our current event tracing tools.

The next experiment was running RTS with preemption where CP work items had higher importance than DP work items. Table 2 shows the results of this experiment.

We get considerably more CP work items processed during a 2 minute run. As can be seen, the number of CP work items that can be processed now levels off near 8,400. The varying figures have to do with the fact that preemption occurs at various levels of completion of DP work items. Notice that approximately 2,800 DP work items still get processed. In order to reduce this preemptive *leaking* further it would be necessary to design the system to preempt running tasks, thus increasing preemption overhead. Notice that the preemption overhead at task boundaries is insignificant up to about 5,000 requests for CP work items in a 2 minute interval. In fact, as in the previous case, with moderate numbers of CP work item requests, RTS runs better than the theory predicts. For real world systems this high level of work item preemption would probably never occur, thereby showing that our method introduces little preemption overhead into the solution. Subsequently, we took the opportunity to run our case study on a Sparcstation 10 and a Silicon Graphics IRIS 4D/440VGXT workstation. We noticed that the mean service time for DP processing and the corresponding standard deviation were an order of magnitude smaller. Although we did not have time to make all the measurements made on the SUN 3/60, the results appear to scale linearly, i.e., standard deviations are still nearly three orders of magnitude less than measured computation times of work items. The Ada compilers we used (and run-time environments) were from Verdix. Though different versions existed on

Table 2. Results of preemption at task boundaries

CP requests	Actual CPs	CP mean	DPs	DP mean	Total time	Percent	Contention overhead
1,001	1,001	9.915	16,380	7.348	133,959	111.63	-11.63
2,001	2,001	9.915	14,972	7.348	129,854	108.21	-8.21
3,001	3,001	9.915	12,775	7.348	123,625	103.02	-3.02
4,001	4,001	9.915	11,078	7.348	121,071	100.89	-0.89
5,001	5,001	9.915	9,375	7.348	118,472	98.73	1.27
6,001	6,001	9.915	7,736	7.348	116,344	96.95	3.05
7,001	7,000	9.915	5,584	7.348	110,436	92.03	7.97
8,001	8,001	9.915	3,590	7.348	105,709	88.09	11.91
9,001	8,330	9.915	2,780	7.348	103,019	85.35	14.15
10,001	8,402	9.915	2,802	7.348	103,915	86.60	13.4
11,001	8,330	9.915	2,780	7.348	103,019	85.85	14.15
12,001	8,398	9.915	2,801	7.348	103,848	86.54	13.46

different platforms, we found the impact of these differences to be negligible in our experiments.

## 6. Proofs of Peer Tasking Behavior

The D-M model assumes that the status of a task whose start time has elapsed can be characterized by two parameters,  $C$  and  $D$ , that represent the computation time ( $C$ ) and the deadline ( $D$ ) by which the task must complete. Furthermore, it is implicitly assumed that the schedulability of a set of tasks  $\{t_i\}$  for  $i = 1 \dots m$  can be determined solely by this information,  $C_i$  (the computation time for task  $t_i$ ), and  $D_i$  (the deadline for task  $t_i$ ) for each task  $t_i$  under the conditions stated in the hypotheses of their theorems. The D-M model rules out data- or state-dependent algorithms in tasks since the task computation times are assumed to be constant. It also rules out pre-emption overhead or scheduling algorithm overhead. There are many real world problems where it is not possible to know the computation times  $C_i$  very accurately for any or all of the tasks  $t_i$ . This can be due to several reasons. In practice what we usually have is an *average* computation time  $A_i$  and a variance measured by a standard deviation  $s_i$ . In many cases  $s_i$  can be significant with respect to  $A_i$ .

To prove the behaviors, we use the following theorem (ref. 1, p. 1503) which was proved for both single and multiple processor machines.

*THEOREM (Dertouzos and Mok, 1989). If a schedule exists that meets the deadlines of a set of tasks whose start times are the same, then the same set of tasks can be scheduled at run time even if their start times are different and not known a priori. Knowledge of the preassigned deadlines and computation times alone is enough for scheduling. One successful run-time scheduling algorithm is the Least Laxity algorithm.*

Let us now consider what it is we need to prove. First, we want to show that if a work item  $w$  can be performed once in time  $C$  by a collection of tasks  $\{t_i\}$  where  $i = 1 \dots m$ , then  $w$  can always be scheduled at run time to meet a deadline  $D = C$  provided there do not exist work items of higher priority to be scheduled when the request for service for this work item is made, and provided no requests for service for a work item of higher priority is made before time  $C$ . This is equivalent to constant computation time for a work item; the constant is the inverse of the throughput (for this type of work item), calculated in work items per second. If it were known that the computation time of a work item was the sum of the computation times of the tasks that accomplish the work item then, of course, the work item computation time

would be constant and there would be nothing to prove. But since computation times in the D-M model are known at the task level and since it is known that the total time it takes for a collection of tasks to execute is not necessarily the sum of the computation times of the individual tasks (recall the theoretical overhead of RMS (ref. 5)) it is not clear without proof that the computation time for a given type of work item is constant.

If we know that a given work item  $w$  can be executed once in time  $C$  then all the tasks in  $w$  would be executed within time  $C$ . Let  $W$  denote the set of tasks that accomplish  $w$ . If we set the start time for each of these tasks to zero (even though they do not all start at time zero, which is permitted in the model) and the deadline for each of these tasks to  $C$ , then it is clear that the task set  $W$  was scheduled in such a way that all the tasks in  $W$  met the common deadline  $C$ . Hence a schedule exists as described in the hypothesis of their theorem. Now applying their theorem, we see that in the future it will always be possible to schedule this task set  $W$  at run time such that the deadline  $C$  can be met, provided, of course, that there does not exist a higher priority work item to schedule at the time the request for service for  $w$  is made and provided no request for service for a higher priority work item is made prior to time  $C$ . Consequently, we have established what we set out to prove. We record this result as

*THEOREM 1. If a work item  $w$  can be scheduled once to execute in time  $C$ , then the set of tasks that executes  $w$  can always be scheduled at run-time to meet a deadline equal to  $C$  provided there is no request for service for a work item of higher priority during this time period.*

Theorem 1 applies to both single and multiprocessor machines since the Dertouzos and Mok theorem is so proved. Also Theorem 1 implies that the variance in the execution time of  $w$  is zero, which never happens in the real world. This is strictly a property of the D-M model.

Next we want to show that for a single processor machine there is no on-line task scheduling overhead for work items that are performed by a collection of peer tasks. This is not to say there is no scheduling overhead, because we must schedule the work items. For example, in our case study, scheduling overhead at the task level defaulted to a dispatch operation of first in first out (FIFO).

We again invoke Dertouzos and Mok's theorem. The Least Laxity algorithm will successfully produce an on-line schedule for a schedulable peer task set  $W$  corresponding to the work item  $w$  given the preassigned deadline  $D$  (which we assign to each task in  $W$ ) and the

work item computation time  $C \leq D$ . For a single processor machine, the Least Laxity algorithm and the Earliest Deadline First (EDF) algorithm are equivalent in that if a task set can be scheduled with one of them then it can be scheduled by the other. Dertouzos and Mok state the EDF algorithm as follows:

*Execute at any time the task whose deadline is closest. Ties are broken arbitrarily.*

Consequently, EDF is an algorithm that will be successful in producing an on-line schedule for the task set  $W$  given the knowledge of the preassigned deadline  $D$  and work item computation time  $C$ . Since the tasks in  $W$  are peer tasks, none have priority over the other, so there is no task level scheduling overhead to assure a particular ordering of the tasks. Also, since all the tasks have the same deadline  $D$ , whatever order they are executed in will satisfy the EDF scheduling criteria. Therefore, we can accept the default scheduling algorithm of the run-time system or operating system with the assurance that this default scheduling will be optimal. In particular, if the system is implemented in the Ada programming language as our test bed is, we control the execution order of the peer tasks merely by the proper use of guards on the task entries and task calls to assure proper logical execution of the task set. We have established

*THEOREM 2. For a single processor machine, there is no on-line task scheduling overhead in the peer tasking theory.*

*COROLLARY. Peer tasking forces optimal task level scheduling by default on a single processor machine.*

The proof of the corollary follows from the fact that whatever order the tasks are executed in satisfies the EDF scheduling criteria and the fact that EDF scheduling is optimal for a single processor machine (ref. 5).

Finally, we want to show that throughput is not degraded during overload. By an arbitrary work load  $A$  we mean a finite sequence of work items  $\{w_i\}$ ,  $i = 1 \dots m$  such that if  $i < j$  then the request to do  $w_i$  occurred prior to the request to do  $w_j$ . We assume that the system receives the sequence  $A$  and reorders it into a sequence  $W$  of work items,  $w_j$ ,  $j = 1 \dots n$ , consisting of all requests received up to the present (so  $n \leq m$ ) and such that if  $i < j$  then  $w_i$  is more important than  $w_j$  or they are of equal importance and the request for  $w_i$  preceded the request for  $w_j$ . We refer to  $W$  as a prioritized work load.

By an overload period we mean a period of time  $P$  that begins at some time  $t_0$  at which a scheduling decision must be made and a prioritized work load  $W$  consisting several work items such that all the work items in  $W$  are requested to finish by  $t_0 + P$  and the computation time of

any member of  $W$  is less than or equal to  $P$ , but  $P < M$  where  $M$  is the sum of the computation times of members of  $W$ . We say that throughput is not degraded during overload if for each overload period  $t_0, P, W$ , the maximum number of work items from  $W$  are completed by  $t_0 + P$  subject to the constraint that no additional requests occur before  $t_0 + P$  and if  $w_1 \in W$  is completed and  $w_2 \in W$  is not completed by  $t_0 + P$ , then either the importance of  $w_1$  was greater than the importance of  $w_2$  or else  $w_1$  and  $w_2$  had the same importance but  $w_1$  was requested prior to  $w_2$ .

First we demonstrate that it is true for a workload consisting of a single work item type, all with equal importance. All work items in  $W$  have the computation time  $C$  (by Theorem 1). Since work item processing time is constant, the system will produce a throughput  $T = P/C$  work items.  $T$  may not be an integral number, so let  $R = T$  truncated to the nearest positive integer. Then there must exist more than  $R$  work items in  $W$ . Consequently, if the system logic is to do the first  $R$  requests for this particular work item, then the maximum number of work items from  $W$  are completed subject to the constraint that if  $w_1, w_2 \in W$  and  $w_1$  is executed but  $w_2$  is not, then  $w_1$  was requested prior to  $w_2$ . Therefore, throughput is not degraded during overload in this case.

Next consider a workload  $W$  consisting of work items of multiple types or importances. Let  $w_1$  be the first work item in  $W$ . Then  $w_1$  has some computation time  $C_1$  and we know that the system can process  $T_1 = P/C_1$  of this type work item during  $P$ . Let  $R_1 = T_1$  truncated to the nearest positive integer. If there are at least  $R_1 + 1$  consecutive work items in  $W$  of the type of  $w_1$ , the system will process  $R_1$  of them during  $P$ . Hence the maximum number of work items from  $W$  subject to the constraint that if  $w_1, w_2 \in W$  and  $w_1$  is processed but  $w_2$  is not, then either  $w_1$  was more important than  $w_2$  or else they have the same importance but the request for  $w_1$  preceded the request for  $w_2$ . In other words throughput will not be degraded during overload.

If there do not exist  $R_1 + 1$  consecutive work items of this type in  $W$ , let  $k_1 \leq R_1$  denote the number of consecutive work items of this type in  $W$ . By Theorem 1, a time of  $k_1 C_1$  will be expended executing these  $k_1$  work items and there will be  $P - k_1 C_1$  time remaining to process the work items in  $W$ . Let  $w_2 \in W$  be a work item in  $W$  following the  $k_1$  work items of the type of work item  $w_1$ . If the computation time  $C_2$  of  $w_2$  is greater than  $P - k_1 C_1$ , then again we have shown no degradation of throughput during overload in this case. If  $C_2 < P - k_1 C_1$ , then by an argument similar to the above, either there will be no throughput degradation during overload or else there

exists a positive integer  $k_2 \leq R_2$  where  $R_2$  is the positive integer obtained by truncating  $T_2 = (P - k_1 C_1)/C_2$ .

We cannot continue this process indefinitely since  $W$  has only finitely many work items. Hence after some  $k$  iterations, there will be more consecutive work items of some type  $w_j$  than can be processed during the remaining time, so, as before, the system will process the maximum number of work items from  $W$  subject to the constraint that if  $w_1, w_2 \in W$  and  $w_1$  is processed but  $w_2$  is not, then either  $w_1$  was more important than  $w_2$  or else they had equal importance but the request for  $w_1$  preceded the request for  $w_2$ . Therefore, throughput is not degraded during overload. We record this result as

*THEOREM 3. There is no throughput degradation during overload with peer tasking.*

Again, Theorem 3 holds on both single and multiprocessor machines.

## 7. Concluding Remarks

The peer task method addresses the “disconnect” between design theory and scheduling theory. This paper has tied together an existing real-time mathematical scheduling model with three new scheduling properties, our prescriptive design guidelines, and a design case study with measurements of how close its real-time behavior approaches what is predicted.

The peer task method offers a scheduling guarantee that is qualitatively different from the type of guarantee offered by urgency based scheduling. The work item throughput guarantee is localized in time and with respect to the set of tasks that accomplishes the work item. Furthermore, this guarantee is independent of state; it will always hold regardless of the load on the system. We call this a hard local guarantee. Urgency based scheduling offers what we call soft global guarantees. By soft we mean that a guarantee is dependent on the state of the system. By global we mean that it applies to all tasks at all times. This is because conventional urgency based scheduling tries to guarantee that all deadlines are met as long as the theoretical overload threshold is not exceeded. The theoretical overload threshold depends on the algorithm employed. For EDF on a single processor machine it is 100 percent processor load. For RMS it is approximately 69 percent for ten tasks (ref. 5). Above these theoretical load levels these guarantees are void and the behavior is not guaranteed.

Our design method rules produce what can be thought of as first ( $1^0$ ), second ( $2^0$ ), and third order ( $3^0$ ) effects. Rule 1 produces  $1^0$  effects because it guides system decomposition for scheduling purposes, namely into

specification level work items. This rule has the greatest overall effect on the behavior of the system. Rule 2 produces  $2^0$  effects at the task (process) level. It guides assignment of objects and functions to tasks. Rule 3 gives  $3^0$  effects because it optimizes the tasks (or processes). It has the least effect on overall system behavior. Although these rules guide, they are not a recipe for all the details of a successful design.

As a system organizing principle for scheduling, the method uses importance. This abstraction is superior to urgency based priority schemes for systems where overload is unavoidable and dynamic reallocation of resources based on user or domain specified semantics is a necessity.

## 8. Further Research

The effective use of the work item concept is closely tied to the capability to implement the abstraction of importance. More research is needed on this abstraction as a scheduling mechanism. A recent survey (ref. 15) on classical scheduling results focuses exclusively on urgency based results. Many of these results are significantly practical in specific domains. But efforts in generalizing these have led to increasingly complex solutions so that the effort to implement, maintain, and modify systems accommodating these outweighs the benefits. No less than a major shift in thinking outside the pale of existing mathematical models may be needed. Whether the approach introduced here of statistically approximating the D-M model may be carried further without falling into the same trap remains to be investigated. Perhaps the D-M model may be augmented, or replaced, in a way that retains significantly practical results yet incorporates more fully the importance concept. This area is one candidate for further research. Besides the development of its theoretical underpinnings, standard services to support its implementation are needed.

Development of such services has begun as seen in user defined or customized scheduling features in MACH 3 (ref. 16). With this facility, users may select and control system resources through their own designation of parameters and embed their own custom scheduling algorithms in the operating system kernel. Additional flexibility and customization of such facilities over what is available here may be needed to make full use of importance based scheduling. This area is another candidate for further research. On the other hand, support facilities for importance may be coming from outside the conventional disciplines of real-time scheduling and operating systems. The technology driven to address

multimedia's and the information highway's needs have significant real-time requirements. The infrastructure being developed to address these may provide a rich set of interfaces to resources that will enable importance based scheduling in LDPARTS, particularly those that have been designed to provide configurable isochronous services. Investigation of the definition and use of such services in a case study using the peer task method is another area for further research.

Finally, more experience in the application of the method is needed. We have discussed the underlying elements of the method, the abstraction of importance and the means to implement it. Separate from these, further investigation in how to use and tailor these guidelines in the initial phases of a project is needed. Preferable to inventing new software packages is the investigation of adapting and integrating existing ones. Since the method is best suited in a "cut and try" project development environment, and given the current trend to rapid prototyping to evaluate and verify system requirements, our approach may be readily adopted. However, rapid prototypes are often built to be discarded after initial evaluation, and the tools used offer little, if any, integration into the remainder of the life cycle. Because of our method's reliance on statistically significant data gathered in the real environment such use of rapid prototyping is not optimal. A better approach is to use a tool such as ROOM (ref. 17) that integrates rapid prototyping into the life cycle. Further research with contemporary tools and paradigms such as objects will help us build in-budget and on-time large scale distributed parallel architecture real-time systems.

## References

1. Dertouzos, M.; and Mok, A.: Multiprocessor On-Line Scheduling of Hard Real-Time Tasks. *IEEE Transactions on Software Engineering*, vol. 15, no. 12, Dec. 1989, pp. 1497–1506.
2. LeLann, G.; and Rivierre, N.: Real-Time Communications over Broadcast Networks: the CSMA-DCR and the DOD-CSMA-CD Protocols. *Rapport de Recherche No. 1863*, Institut National de Recherche en Informatique et en Automatique (in English), Mar. 1993.
3. Proceedings of the Workshop on Large, Distributed, Parallel Architecture, Real-Time Systems. IDA Document No. D-1425, D. Fife et al., eds., Mar. 1993.
4. Liu, J.: Issues in Distributed Real-Time Systems. *Proceedings of the Workshop on Large, Distributed Parallel Architecture, Real-Time Systems*, Institute for Defense Analyses Document D-1425, July 1993.
5. Liu, C.; and Layland, J.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, vol. 20, no. 1, Jan. 1973, pp. 46–61.
6. Kavi, K. M.; and Yang, Seung-Min: Real-Time Systems Design Methodologies: An Introduction and a Survey. *Journal of Systems Software*, vol 18, 1992, pp. 85–99.
7. Briand, L.: Ada Real-Time Systems and Basic Priority Inheritance. *ACM Ada Letters*, vol. 14, no. 3, May–June 1994.
8. Clark, R.: Scheduling Dependent Real-Time Activities. In *Decentralized Real-Time Scheduling*, Rome Air Development Center Technical Report No. RADC-TR-90-182, Aug. 1990.
9. Howes, N.: Real-Time Ada Design Methodologies and their Impact on Performance. *IDA Paper No. P-2488*, June 1991.
10. Young, S.: *Real Time Languages: Design and Development*. Ellis Horwood Limited, England, 1982.
11. Nielsen, K.; and Shumate, K.: *Designing Large Real-Time Systems with Ada*. McGraw-Hill, New York, 1988.
12. Sanden, B.: Entity-Life Modeling and Structured Analysis in Real-Time Software Design—A Comparison. *Communications of the ACM*, vol. 32, no. 12, Dec. 1989, pp. 1458–1466.
13. Howes, N.: Toward a Real-Time Ada Design Methodology. *ACM Proceedings Tri-Ada '90*, Dec. 1990.
14. Buhr, R.: *System Design with Ada*. Prentice-Hall, Englewood Cliffs, N.J., 1984.
15. Stankovic, J.; Spuri, M.; Di Natale, M.; and Buttazzo, G.: Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, June 1995.
16. Loeper, K., ed.: *Mach 3 Kernel Interface*. Open Software Foundation and Carnegie Mellon University, Boston, Mass., Mar. 1992.
17. Selic, B.; Gullekson, G.; McGee, J.; and Engelberg, I.: ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. *Proc. 5th International Workshop on CASE*, Montreal, Canada, 1992.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE October 1995		3. REPORT TYPE AND DATES COVERED Technical Memorandum
4. TITLE AND SUBTITLE  Real-Time Design with Peer Tasks			5. FUNDING NUMBERS  233-02-07	
6. AUTHOR(S) Andre Goforth, Norman R. Howes,* Jonathan D. Wood,* and Michael J. Barnes†				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)  Ames Research Center Moffett Field, CA 94035-1000			8. PERFORMING ORGANIZATION REPORT NUMBER  A-950095	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)  National Aeronautics and Space Administration Washington, DC 20546-0001			10. SPONSORING/MONITORING AGENCY REPORT NUMBER  NASA TM-110367	
11. SUPPLEMENTARY NOTES Point of Contact: Andre Goforth, Ames Research Center, MS 269-4, Moffett Field, CA 94035-1000 (415) 604-4809 *Institute for Defense Analyses, Alexandria, Virginia    †Recom Technologies, Moffett Field, California				
12a. DISTRIBUTION/AVAILABILITY STATEMENT  Unclassified — Unlimited Subject Category 59			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)  We introduce a real-time design methodology for large scale, distributed, parallel architecture, real-time systems (LDPARTS), as an alternative to those methods using rate or dead-line monotonic analysis. In our method the fundamental units of prioritization, <i>work items</i> , are domain specific objects with timing requirements (deadlines) found in user's specification. A work item consists of a collection of tasks of equal priority. Current scheduling theories are applied with artifact deadlines introduced by the designer whereas our method schedules work items to meet user's specification deadlines (sometimes called end-to-end deadlines).  Our method supports these scheduling properties. Work item scheduling is based on domain specific importance instead of task level urgency and still meets as many user specification deadlines as can be met by scheduling tasks with respect to urgency. Second, the minimum (closest) on-line deadline that can be guaranteed for a work item of highest importance, scheduled at run time, is approximately the inverse of the throughput, measured in work items per second. Third, throughput is not degraded during overload and instead of resorting to <i>task shedding</i> during overload, the designer can specify which work items to shed. We prove these properties in a mathematical model.				
14. SUBJECT TERMS  Real time, Scheduling, Methodology			15. NUMBER OF PAGES 19	
			16. PRICE CODE A03	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT	